

# Supplementary Information and Results: Bioinformatics Pipeline using JUDI: *Just Do It*

Soumitra Pal      Teresa M. Przytycka

## S1 Introduction

Large scale analysis of data in Bioinformatics requires many software to be executed in a pipelined fashion where output of a upstream stage is fed as input to a downstream stage. Generally each such software takes a considerable amount of computing resources such as CPU time and memory. If the pipeline needs to be executed multiple times, may be due to input or parameter variations, it is desired that only the bare minimum stages that are affected by the variations are re-executed. The rest should be reused from the previous execution. To address such a requirement, many workflow management systems (WMS) have been developed. One of the earliest and widely used system is Make [S5]. Each stage of the pipeline is specified as a rule of commands in Make to generate the output(s) from the input(s). Make automatically figures out the interdependency among the rules and by using a directed acyclic graph (DAG) of the rules executes the commands in a suitable order such that all inputs are already available before a command is executed. Moreover, if all inputs are unchanged, Make skips the execution of that command.

Make has a steep learning curve, specifically for the biologists. Several newer WMS have been developed to specify Make rules in simpler languages, e.g., Snakemake [S2] and Nextflow [S1] use simplified Python and Groovy, respectively. These WMS can also schedule the pipeline tasks efficiently on multiprocessor environments by exploiting the DAG of rules.

However, these WMS require significant effort in scripting to ensure optimal execution of the stages when the pipeline needs to be executed multiple times due to changes in different parameters such as thresholds, algorithmic methods, and hyper-parameters of machine learning algorithms. Generally these changes affect only a part of the pipeline leaving some scope for resource saving by fresh execution of the affected parts only. Though Snakemake and Nextflow enable adhoc use of wildchar parameters for easy scripting, a systematic handling of parameters is lacking in the literature. Moreover, these systems focus on each stage of the pipeline separately and the lack of information sharing across stages makes plug-and-play of stages difficult. For example, if a file generated by one stage is used as input in multiple stages downstream then the details of the file need to be re-specified unnecessarily in each downstream stage making it tedious when there are several parameters.

We develop *JUDI* on top of a Python based build system, *DoIt* [S4], to systematically handle the issue of parameter settings using the principles of Data Base Management Systems (DBMS). By abstracting each file and task in a pipeline stage with an associated parameter database JUDI additionally enables true plug-and-play of pipeline stages. The novel ideas in JUDI not only simplify pipeline scripting but also reduce script size significantly.

## S2 Parameter Database

The basic unit of execution in DoIt [S4], the underlying workflow management in JUDI, is a task roughly equivalent to a collection of rules in Make [S5]. Like other workflow systems JUDI uses the dependency among tasks based on their input and output files. However, the novelty of JUDI is a consolidated way of capturing the variability under which the pipeline being build could possibly be executed. This variability could be the parameters to the software used in the pipeline stages or could be some new parameters introduced for the overall pipeline. JUDI stores this variability in a simple table which contains one column for each parameter where each row indicates a value of the parameter. The database is populated one parameter at a time by taking a Cartesian product of the list of categorical values the parameter can take with the database constructed so far. Multiple parameters are populated through a table which need not necessarily be the Cartesian product of the parameters represented by the columns.

Table S1 illustrates parameter databases in JUDI using two example databases. Database A has two parameters: 1) *sample* having four values {100, 101, 102, 103} could be representing sequencing data from four individuals, and 2) *group* having two values {1, 2} representing the end of paired-end sequencing data. Thus, the database table has two columns representing the two parameters and  $4 \times 2 = 8$  rows representing the Cartesian product of the two parameters. The four samples themselves could have been from three patients, as shown in database B where the first two patients have one sample each and the third patient has two samples. Thus the overall database could have represented the Cartesian product of 1) a database of two variables: (patient, sample) with four rows (P1, 1), (P2, 1), (P3, 1) and (P3, 2) and the single parameter, group, with two values {1, 2}.

Table S1: Examples of parameter database									
<div>sample    group</div>		<div>patient    sample    group</div>							
A	100	1	B	P1	1	1	<div>patient    sample</div>		<div>group</div>
	100	2		P1	1	2	P1	1	
	101	1		P2	1	1	P2	1	
	101	2		P2	1	2	P3	1	
	102	1		P3	1	1	P3	1	2
	102	2		P3	1	2	P3	2	
	103	1		P3	2	1			
	103	2		P3	2	2			

## S3 JUDI File

Each JUDI file is associated with a parameter database and hence does in fact represent a collection of physical files each corresponding to one row of the parameter database. When associated with example database A, a JUDI file *reads* could represent the eight FASTQ *file instances*: {100\_1.fq, 100\_2.fq, 101\_1.fq, 101\_2.fq, 102\_1.fq, 102\_2.fq, 103\_1.fq, 103\_2.fq}. The mapping from the rows of the parameter database to the physical path of the file instances is stored by an extra column for path in the parameter database.

A by-product of clubbing several physical files as a JUDI file is that if those files are defined in an upstream task, any number of downstream tasks can refer to them by a single name instead of

referring to or re-specifying each of the physical files. Thus, both inputs and targets are collections of files which could be treated as input and output terminals of hardware modules, and hence providing better plug-and-play.

## S4 JUDI Task

Each JUDI task has four components: 1) *inputs*, 2) *targets*, 3) *actions* and 4) *parameter database* where the first three components are similar to those in DoIt with the difference that each element of inputs and targets is a JUDI file instead of a physical file. Like a JUDI file, a JUDI task represents a collection of *task instances* each corresponding to a row of the parameter database. For example, a JUDI task with parameter database A, an input ‘reads’ of the example above and a target ‘bam’ could align each of the eight FASTQ files to a reference genome and generate eight BAM files where the alignment software and its arguments are specified through one of the actions for the task.

## S5 Determining File Instances for a Task Instance

Consider a JUDI task  $T$  with parameter database  $D_T$  and a JUDI file  $F$  (input or target) with parameter database  $D_F$  with common parameters  $X$ . In many cases like the example with ‘reads’ and ‘bam’ above,  $D_F = D_T$ . Otherwise, there could be four possible scenarios as shown in Fig. S1. First, when  $D_T$  has an extra parameter  $Y$ , for each  $y \in Y$  instance  $(x, y)$  of  $T$  gets the same instance  $x$  of  $F$ ,  $\forall x \in X$ . Second, when  $D_F$  has an extra parameter  $Y$  instance  $x$  of  $T$  gets a list of all instances  $\cup_y \{(x, y)\}$  of  $F$ . Third, when  $D_F, D_T$  have same parameters but row  $x$  of  $D_F$  is missing in  $D_T$ , it does not create a problem as far as task instances are concerned. Fourth and final, when row  $x$  of  $D_T$  is missing in  $D_F$ , opposite to third case, the instance  $x$  of  $T$  is undefined and hence should be dealt as an error.

Thus, to determine the instances of  $F$  accessed in an instance of  $T$  the following sequence of basic database operations are performed. Let  $D_{T,F}$  denote the *left outer join* between  $D_T$  and  $D_F$  on common columns  $X$  and for each (unique) row  $r$  of  $D_T$ , let  $D_{T,F}^r$  denote the projection  $\Pi_r(D_{T,F})$ . Then instance  $r$  of  $T$  gets all file instances in  $D_{T,F}^r$ . Along with the physical path information,  $D_{T,F}^r$  also contains the settings of extra parameters (if any) in  $D_F$  and could be used by any summarizing task.

## S6 Implementation of JUDI using DoIt

JUDI is available as a Python library and any JUDI pipeline first populates a global parameter database using the function `add_param` to avoid repeated local definition in each task. A task is a class derived from the base class `Task` and should have four class variables: 1) `mask`, 2) `inputs`, 3) `targets`, 4) `actions`. Each of `inputs` and `targets` is a python dictionary where each key names an input or target terminal (see Section S3) of the task and has as value an object of class `File`. The object can be newly created using the class constructor or can refer to a `File` object created before. For example, if an input terminal of a task A is connected to the target terminal `t` of another task B, then A can access the corresponding file(s) directly by `B[t]`. In this way, the referring task A need not repeat the definition of the file.

If the class variable `mask` is set in a task then it represents the list of parameters from the global parameters database that are not applicable to the current task and the masked parameter database for the current task is created accordingly. Otherwise the task parameter database is assumed to be the same as the global parameter database. The actions for a task are specified by

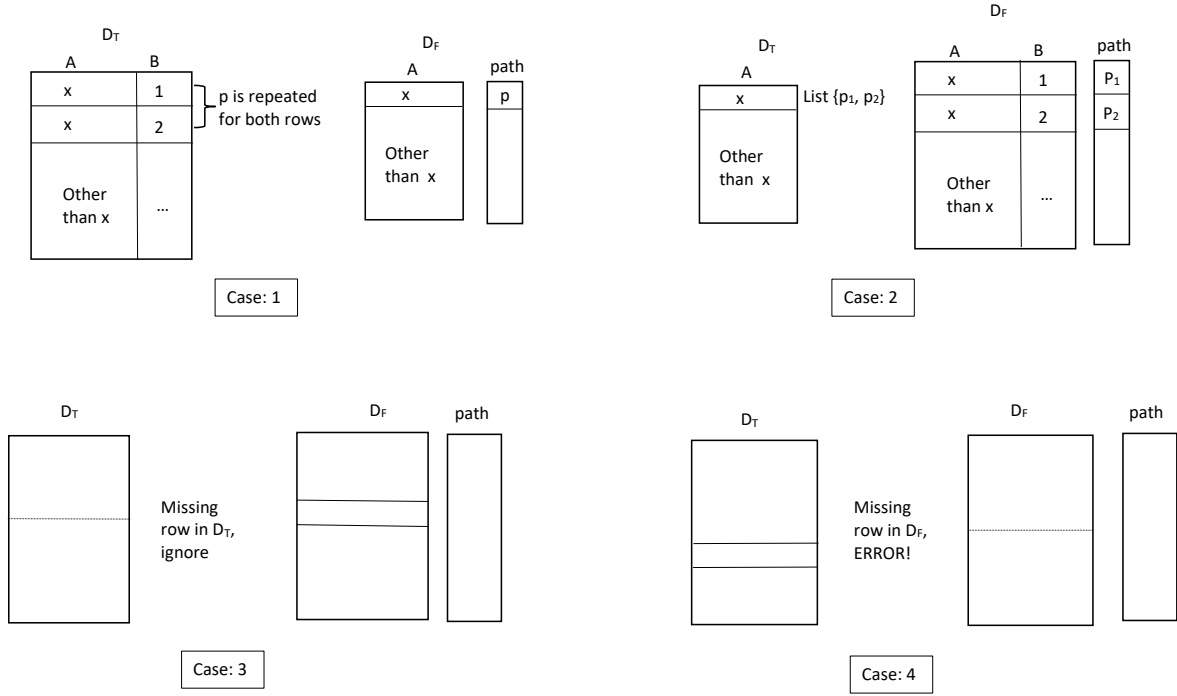


Figure S1: **Operation on parameter databases for a task and a file.**

the python list `actions` of tuples ( $fun, args$ ) where  $fun$  could be a Python string denoting the command line specification of the action with placeholders `{}` which are replaced by the list  $args$  of values similar to Python `format` function with following exception: if a value is “\$x” then the placeholder is replaced by a string containing the list of paths of the file instances in  $D_{T,x}^r$  for the task instance  $r$ . A  $fun$  could also be a Python function and  $args$  could additionally have a value “#x” which JUDI replaces by a pandas DataFrame for  $D_{T,x}^r$  to pass to  $fun$ .

## S7 A Toy Example from Snakemake Paper

We use a slightly modified version of the pipeline used in the Snakemake paper [S2] to illustrate the ideas used in JUDI. The pipeline has four stages as shown in Figure S2. The first three stages are same as in the Snakemake example. In the first stage, each of the eight FASTQ files, one for each combination of 4 samples and 2 groups of pair-end reads, as mentioned in Section S2 is aligned to an intermediate file. The intermediate files for the two groups of pair-end for each sample is converted to a BAM file in the second stage of the pipeline. For each sample, the third stage generates a table containing the coverage information. The fourth stage is different from the Snakemake example in the sense, first the coverage tables of all samples are merged into a single consolidated coverage table and then this combined information is plotted, unlike the fourth stage in Snakemake example where the coverage plot is generated separately for each sample. Fig. S3 shows the pipeline from Fig. S2 using JUDI tasks and files along with their parameter databases

Listing 1 shows the Python script `dodo.py` for the pipeline implemented using JUDI. When command line `doit -f dodo.py` is executed, JUDI python library creates the task instances which

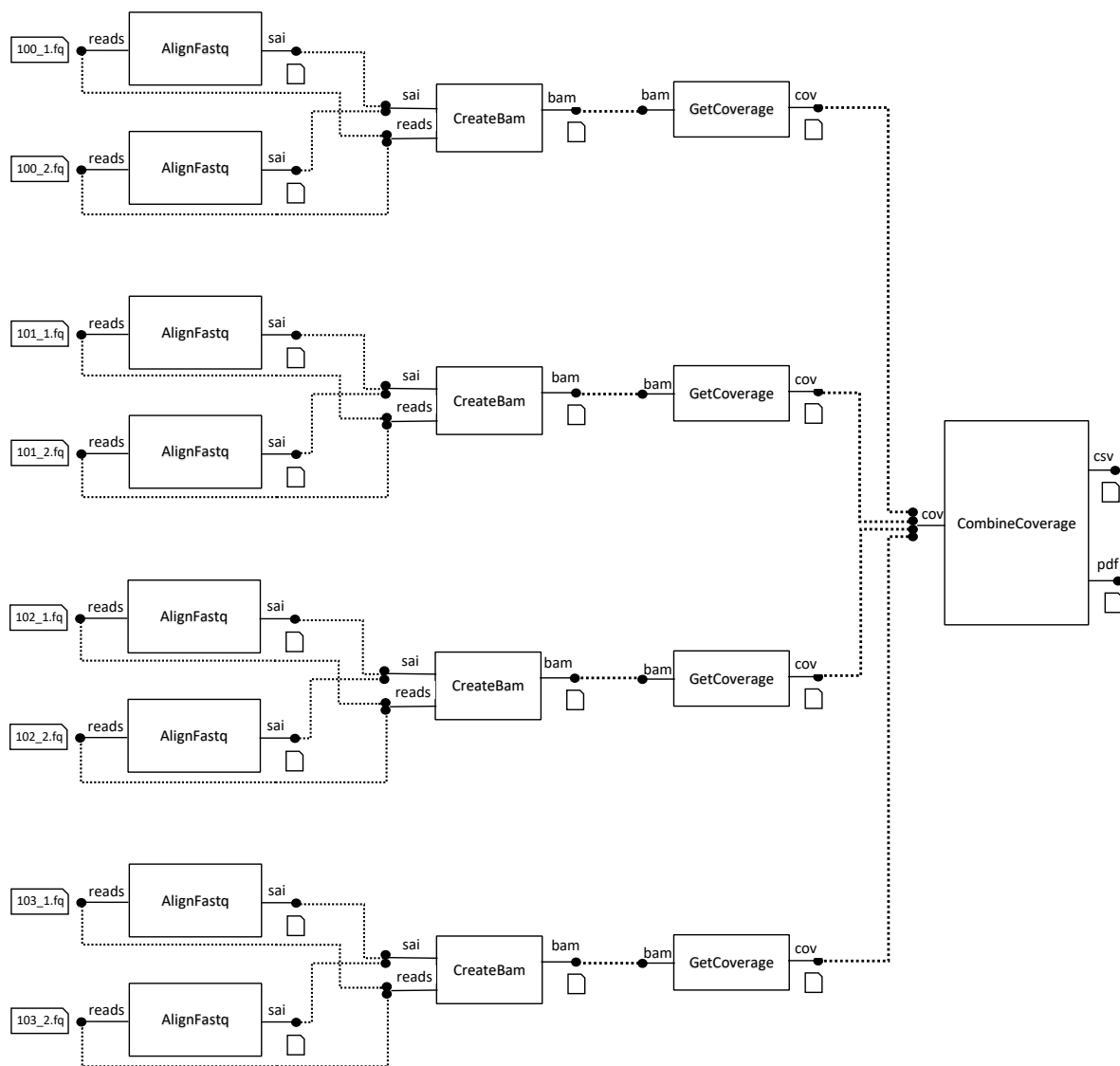


Figure S2: **An example pipeline.** We slightly modified pipeline used in the Snakemake paper [S2] to illustrate usefulness of JUDI.

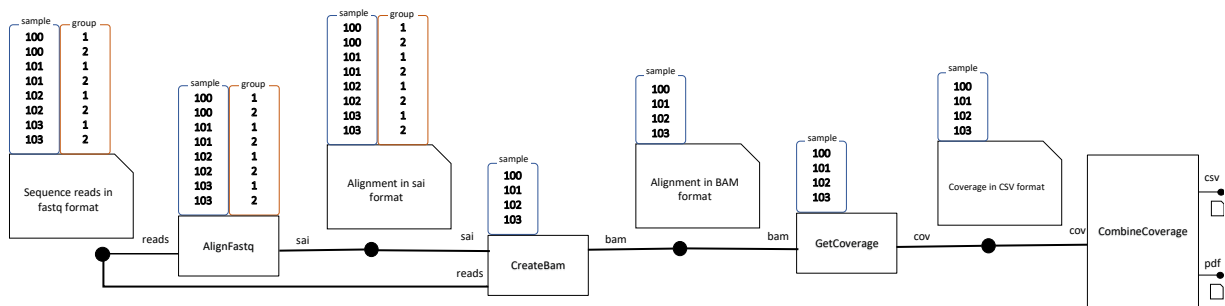


Figure S3: **Example pipeline in Fig. S2 visualized using JUDI concepts.**

are executed by the DoIt engine. More details of JUDI and DoIt can be found in their users manuals, [pydoit.org](http://pydoit.org) and [judi.readthedocs.io](http://judi.readthedocs.io), respectively.

The usefulness of JUDI can be illustrated with two tasks in Listing 1. CreateBam has only one parameter ‘sample’ whereas its both input files ‘reads’ and ‘sai’ have both parameters ‘sample’ and ‘group’. For each sample, the physical paths of sai and fastq files for both pair-end reads are passed to the aligner **bwa sampe** automatically by JUDI through the argument substitutions **\$sai** and **\$reads**. Similarly, CombineCoverage has no parameter but its input **cov** has a parameter **sample**. The JUDI library function **combine\_csvs** merges the coverage files for all samples using  $D_{T,cov}^r$  and  $D_{T,csv}^r$  (see Section S5) provided by argument substitutions **#cov** and **#csv**, respectively, where  $T$  denotes CombineCoverage.

## S8 A Real Example: Co-SELECT

We conceived the idea of JUDI while developing Co-SELECT [S3] for analyzing HT-SELEX data of transcription factor (TF) DNA binding. The tool analyzed 5 *rounds* of data from 83 *TF experiments* for 3 *families* by dividing the sequencing reads into two *populations* to find statistically significant shape-strings which were enriched at 5 possible *thresholds* in both populations. Co-SELECT also analyzed *control experiments* taking cross population data for two TFs in two different families.

The implementation of the Co-SELECT pipeline requires about 150 DoIt tasks with overall 2100 lines of Python code. For example, the stage of the pipeline where the oligos from a selection are divided into two categories: motif-containing (referred here as foreground, or fg in short) and motif-free (referred here as background, or bg in short) could be roughly implemented using only DoIt as shown in Listing 2 which takes about 17 lines in addition to the function definition and documentation lines.

The same stage of the pipeline can be implemented using JUDI on top of DoIt as shown in Listing 3 which takes only 3 lines in addition to the function definition and documentation lines, as most of the code due to the parameters are already encapsulated in the parameter database associated with the task. Thus, using JUDI the 150 tasks could be implemented using  $150 \times 3 = 450$  lines. This gives about 5 times reduction in scripting.

In general, for a typical pipeline with  $n$  parameters DoIt takes about 1 (for task definition) +  $n$  (for  $n$  **for** loops) + 7 (for task instance generator) =  $n + 8$  lines of code. Using JUDI on top of DoIt the same pipeline can be implemented using 1 (for task definition) + 4 (task class variables) = 5 lines of code. This gives an  $O(n)$ -factor improvement. This significantly reduces the effort required to maintain the code. Moreover, the functions like **combine\_csvs** further reduce the effort in summarizing results.

## S9 Executing Pipeline in Multiprocessor Environments

To speed up the execution of a pipeline built using JUDI by utilizing multiple CPUs available in the modern processors, DoIt can be invoked by **doit -n N -f dodo.py** where  $N$  denotes the maximum number of independent tasks that DoIt should execute simultaneously.

When the task actions are specified as command strings, the tasks can be executed in a high performance clustering environment by prefixing the command string by a cluster specific blocking command (for example **srn** in Slurm [S6]).

We tested this solution in the Biowulf (<http://hpc.nih.gov>) cluster available at the NIH. We modified the JUDI script for the example pipeline given in the main manuscript. For example, the action for the task for aligning the reads was modified (shown in red) as in Listing 4 where the

Listing 1: **dodo.py: JUDI script for the pipeline in Fig. S3**

---

```

1 from judi import File, Task, add_param, combine_csvs
2
3 add_param('100 101 102 103'.split(), 'sample')
4 add_param('1 2'.split(), 'group')
5
6 REF = 'hg_refs/hg19.fa'
7 path_gen = lambda x: '{}-{}.fq'.format(x['sample'], x['group'])
8
9 class AlignFastq(Task):
10     inputs = {'reads': File('orig_fastq', path = path_gen)}
11     targets = {'sai': File('aln.sai')}
12     actions = [('bwa aln {} {} > {}'.format(REF, '$reads', '$sai'))]
13
14 class CreateBam(Task):
15     mask = ['group']
16     inputs = {'reads': AlignFastq.inputs['reads'],
17              'sai': AlignFastq.targets['sai']}
18     targets = {'bam': File('aln.bam', mask = mask)}
19     actions = [('bwa sampe {} {} {} | samtools view -Sbh - | samtools
20                 sort -> {}'.format(REF, '$sai', '$reads', '$bam'))]
21
22 class GetCoverage(Task):
23     mask = ['group']
24     inputs = {'bam': CreateBam.targets['bam']}
25     targets = {'cov': File('cov.csv', mask = mask)}
26     actions = [('echo val; samtools rmdup {} - | samtools mpileup - |
27                 cut -f4) > {}'.format('$bam', '$cov'))]
28
29 class CombineCoverage(Task):
30     mask = ['group', 'sample']
31     inputs = {'cov': GetCoverage.targets['cov']}
32     targets = {'csv': File('combined.csv', mask = mask),
33              'pdf': File('pltcov.pdf', mask = mask, root = '.')}
34     actions = [(combine_csvs, ['#cov', '#csv']),
35                ("""echo "library(ggplot2); pdf('{}')
36                  ggplot(read.csv('{}'), aes(x = val)) +
37                  geom_density(aes(color = factor(sample)))"\
38                  | R --vanilla""", ['$pdf', '$csv'])]

```

---

Listing 2: Task “partition” using DoIt alone

---

```

1 def task_partition():
2     """ Partition aptamer sequences into motif-containing (fg) and motif-
        free (bg) based on distance from MOTIF """
3     for family in families:
4         for tf in family.tfs:
5             for exp in tf.experiments:
6                 for cycle in exp.cycles:
7                     seq_file = "%s/%s" % (orig_data_dir, getSequenceFile(family,
                        tf, exp, cycle))
8                 for motif in tf.motifs:
9                     fg_file = "%s/%s" % (top_data_dir, getContextFile(family,
                        tf, exp, cycle, motif, 'fg'))
10                    bg_file = "%s/%s" % (top_data_dir, getContextFile(family,
                        tf, exp, cycle, motif, 'bg'))
11                    ensure_dir(fg_file)
12                    ensure_dir(bg_file)
13                    yield {
14                        'name'      : ':'.join([seq_file, motif]),
15                        'actions'   : [(partition_aptamers, [seq_file, motif,
                        fg_file, bg_file])],
16                        'file_dep'  : [seq_file],
17                        'targets'   : [fg_file, bg_file],
18                        'clean'     : True,
19                    }

```

---

Listing 3: Task “partition” using JUDI along with DoIt

---

```

1 class Partition(Task):
2     """ Partition aptamer sequences into motif-containing (fg) and motif-
        free (bg) based on distance from MOTIF """
3     inputs = {'seq': File('seq.fastq')}
4     targets = {'fg': File('fg.txt'), 'bg': File('bg.txt')}
5     actions = [(partition_aptamers, ['$seq', '#motif', '$fg', '$bg'])]

```

---

Listing 4: Modification of command string to execute task in HPC environment

---

```

1 class AlignFastq(Task):
2     inputs = {'reads': File('orig_fastq', path = path_gen)}
3     targets = {'sai': File('aln.sai')}
4     actions = [('srun -n 1 -N 1 -c 1'+bwa aln {} {} > {}', [REF, '
    $reads', '$sai'])]
```

---

command line arguments for the number of nodes (-N), the number of tasks (-n) and the number of CPUs per task (-c) are set accordingly. Though `srun` is a blocking call, the desired parallel execution of multiple tasks is achieved by invoking `doit` with option `-n`.

For HPC systems with other workload managers (e.g. `qsub`) the same can be achieved using the interactive/blocking command line options (e.g. `-I -x`).

For task actions specified as python callables the same can be achieved by moving the function call to standalone python script and calling the new script as a command line string from the task actions as shown in the example above.

## S10 Discussion

In this paper, we introduced a workflow management system with a novel way of handling parameter settings and file path specifications with the motto: define once, reuse many times. We have implemented our ideas using an existing Python based build system DoIt [S4] mainly for two reasons: 1) DoIt does not require to learn any new language in addition to Python, and 2) it is more flexible for implementing our ideas quickly. However, our ideas can also be implemented in other WMS such as Snakemake [S2] and Nextflow [S1]. Though for the simple example in Fig. S3 JUDI takes almost same number of lines as Snakemake, for a larger pipeline as in [S3] JUDI requires far less scripting. For example, the file name expansion due to the masked parameter ‘group’ needed hard-coding in lines 11-12, Listing 1 of [S2], imagine the effort required if there were more masked parameters and one or more parameters had a relatively large number of possible values!

There are a few features in Snakemake [S2] and Nextflow [S1] which are not implemented in JUDI which could be easily implemented in a future version. One such example is to support temporary intermediate files which are not saved across invocation of the pipeline. Nextflow also provides an advanced way of handling files using streams, here we confined on file.

Finally our solution can be extended to have graphical user interface. When combined with interfaces like iPython Notebooks, JUDI could facilitate reproducible research by providing a way to explore unpublished parameter settings.

## Supplementary References

- [S1] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35:316–319, April 2017.
- [S2] Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, October 2012.

- [S3] Soumitra Pal, Jan Hoinka, and Teresa M. Przytycka. Co-SELECT reveals sequence non-specific contribution of DNA shape to transcription factor binding in vitro. *Nucleic Acids Research*, 47(13):6632–6641, July 2019.
- [S4] Eduardo Schettino. DoIt Automation Tool. *URL <http://pydoit.org>. Online*, 2008.
- [S5] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: A Program for Directed Recompilation : GNU Make Version 3.81*. Free Software Foundation, 2004.
- [S6] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pp. 44–60, Berlin, Heidelberg, 2003. Springer.